Applying Bees Algorithm to Tetris

DAANG, JESSA, Ateneo de Davao University
JOVER, GREGG MEYRICK, Ateneo de Davao University

Abstract: This paper presents a different method for adjusting weights of feature functions in the Tetris evaluation function. Using Bees Algorithm (BA), the number of rows cleared was measured to be higher. Results show much agreement with theoretical predictions and significant improvement over previous efforts by Particle Swarm Optimization (PSO). The work presented here has profound implications for future studies of building Tetris agents and may one day help solve the problem of Tetris being NP-Complete.

Keywords: Tetris, Tetris Agent, Bees Algorithm

1. INTRODUCTION

1.1 Background of the Study

In Computer Science, there are two classes of computational problems that researchers are concerned with the search for solutions: P-problems and NP-problems. P-problems (Polynomial-time problems) are problems whose solutions can be easily known. NP-problems are problems whose solutions could not be acquired in a reasonable amount of time. Their solutions can only be valid when tried and tested. One good example of an NP-problem is the creation of an intelligent agent in Tetris.

Tetris is a puzzle computer game originally developed by Alexey Pajitnov. An intelligent agent for Tetris is program whose goal is to play the game in the most optimal way possible. By definition of an NP-problem, the performance of the agent can only be assessed when it has finished playing the game. Since we only know the quality of the agent when it has finished playing the game, it can be difficult to know the solution beforehand.

Moreover, the difficulty and complexity on creating a Tetris agent is documented in several studies. One interesting thing to note is that one cannot win at Tetris in the sense that one cannot play indefinitely [1]. There exists a sequence of Tetrominoes that will always lead to the termination of the game. With 2^{60} states [2] (On a standard 10 x 20 board and seven pieces) and 5.6×10^{59} board configurations, finding a strategy to maximize the average score is an NP-Complete problem even when sequences of pieces are known in advance [3][4].

These properties made researchers consider Tetris as a benchmark optimization problem. Some Tetris agents were developed through Swarm-based optimization (SOA) algorithms, such as Ant Colony Optimization (ACO), Genetic Algorithm (GA) and Particle Swarm Optimization (PSO). Other agents were developed outside SOA principles, such as Reinforcement Learning (RL), Hand-tuning, General Purpose Optimization, Harmony Search Algorithm (HSA) and Evolutionary Algorithms.

Moreover, the study discussed SOA algorithms as feasible means in developing Tetris agents. One of the combinatorial optimization algorithms recently developed is the Bees Algorithm (BA), a new Swarm Intelligence approach, which is inspired by the natural honey bees. Artificial bees are used as agents to communicate with each other and with the environment.

1.2 Problem Statement

The proponents will investigate BA in developing a Tetris agent. In addition, the study seeks to answer the following questions:

(1) How are weight parameters adjusted using the algorithm?
(2) What are the necessary features to be used in evaluating a game board?

1.3 Objectives

The main objective of the study is to implement BA in a Tetris agent and generate reasonable results. In addition, the study desires to accomplish the following objectives:

(1) To identify the features to be considered in order to generate better weights.
(2) To explain how the optimal weight parameters are generated using the algorithm.
(3) To present a Tetris program, showing the utility of the proposed algorithm.

1.4 Significance of the Study

The proposed approach in the study could contribute to the Bees Algorithm and to the design of intelligent Tetris agents. First and foremost, the study could further provide development and applications for future studies, in addition to serving as a foundation for game-based research. It could also enhance the capacity of the algorithm to handle large-scale problems, while providing good solutions. In essence, the said problem is worth the time and effort to be spent studying on it.

1.5 Scope and Limitations

The proponents will conduct a study on how BA can be implemented in generating optimal weight parameters for the game. The study will only consider six features: landing height, eroded piece cells, row transitions, column transitions, holes and wells. The study also considers Tetris boards of varying dimensions, starting from 4 x 20 until 8 x 20 board dimension. The study does not consider Tetris boards beyond the dimension specified, due to time constraints in acquiring the results for the optimal weights.

2. REVIEW OR RELATED LITERATURE

2.1 Tetris

Tetris is a popular video game programmed by Alexey Pajitnov in June 6, 1984 while working in Dorodnicyn Computing Center of the Academy of Science of the USSR in Moscow [5]. It is considered to be one of the most popular and successful games in the market. The success of Tetris led to the creation of other Tetris variants with slightly different game play. Tetris and other variants basically have two main components: the game board and the game pieces called Tetrominoes.

In a Standard Tetris game, the board dimension is set to 10 x 20 and there are seven Tetrominoes as shown in Figure 1, where the Tetrominoes are I, J, L, O, S, T and Z. Each Tetromino varies in its ability to clear rows. Tetrominoes I, J and L are capable of clearing three lines in a row, while Tetromino I can clear four rows.

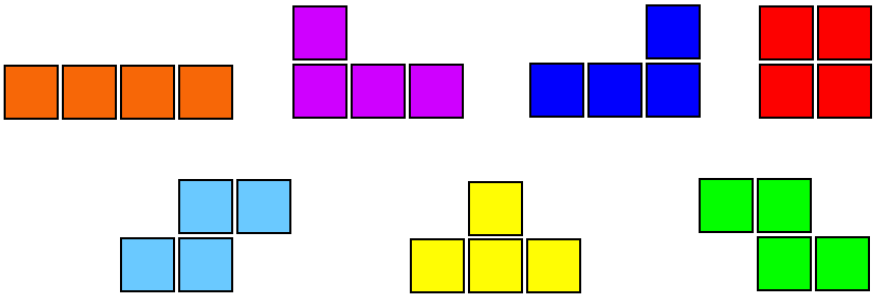

Figure 1. The Seven Tetrominoes Used in Tetris

In Tetris, Tetrominoes fall from the game board one at a time and aside from the current Tetromino being manipulate, an advance view of the next Tetromino is provided to the player in order to manipulate and position the current Tetromino in the game board, such that one or more gapless rows of block is created. When this happens, the gapless row is cleared and all existing blocks above that row descend n units, where n is the number of rows cleared, between 1 and 4. Manipulating a Tetromino can be done in two ways: by doing a rotation (rotate the Tetromino 90 degrees) or a translation (moving the Tetromino sideways) when the Tetromino has not reached the bottom of the board yet, at which case it fixes itself into position.

Unlike most games however, there is no winning condition for Tetris, which means that the game continues until the stack of Tetrominoes disallow the entry of succeeding Tetrominoes. The only goal in the game is to be able to clear as many rows as possible for better and longer gameplay. As a result, staying in the game solely depends upon the number of rows cleared.

2.1.1 Tetris Evaluation Function and Feature Functions

Designing a good Tetris agent comes down to building a good evaluation function [3]. All Tetris agents rely on an evaluation function which is normally a linear summation of all feature functions along with their corresponding weights. The state-evaluation function is shown at Equation 1

$$V(s) = \sum_{i=1}^{n} w_i f_i(s)$$

Formula 1. The State-Evaluation Function

where s is the state (board configuration), w_i represents the weights of each feature function $f_i(s)$ and $f_i(s)$ is a function that maps a state (board configuration) to a real value. There are other combinations of evaluation functions such as an exponential combination proposed by Bohm et al. [6] where they considered the feature functions powered to given constants, and a fast forward neural network proposed by Langenhoven et al. [7] where the value of each feature function is an input node.

The purpose of the evaluation function is to reduce the overall number of game states into a smaller set of features. These feature functions represent the important properties when evaluating a given board. Overall, there are 32 feature functions that could be used in Tetris as shown in Figure 2.

Feature	Description
Max height	Maximum height of a column
Holes	Number of empty cells covered by a full cell
Column height	Height of each column
Column difference	Height difference between each pair of adjacent columns
Landing height	Height where the last Tetromino is added
Cell transitions	Number of full to empty or empty to full cell transitions
Deep wells	Sum of well depths, except for wells with depth 1
Embedded holes	Sort of weighted sum of holes (not precisely documented)
Height differences	Sum of the height differences between adjacent columns
Mean height	Mean height of columns
Δ max height	Variation of the maximum column height
Δ holes	Variation of the hole number
Δ height differences	Variation of the sum of height differences
Δ mean height	Variation of the mean column height
Removed lines	Number of lines completed at the last move
Height weighted cells	Full cells weighted by their height
Wells	Sum of the depth of the wells
Full cells	Number of occupied cells on the board
Eroded piece cells	(Number of rows eliminated in the last move) X (Number of bricks eliminated from the last piece added)
Row transitions	Number of horizontal cell transitions
Column transitions	Number of vertical cell transitions
Cumulative wells	$\sum_{w \in wells}^{n} (1 + 2 + \cdots + depth(w))$
Min height	Minimum height of a column
Max - mean height	Maximum column height - Mean column height
Mean - min height	Mean column height - Minimum column height
Mean hole depth	Mean depth of holes
Max height difference	Maximum difference of height between two columns

Adjacent column holes	Number of holes, where adjacent holes in the same column count only once
Max well depth	Maximum depth of a well
Hole depth	Number of full cells in the column above each hole
Rows with holes	Number of rows having at least one hole
Pattern diversity	Number of different transition patterns between adjacent columns

Figure 2. Overall Tetris Feature Functions [3]

This study will only consider six feature function that were introduced by Dellacherie [5] and these are the Landing Height, Eroded Piece Cells, Row Transitions, Column Transitions, Holes and Wells.

2.1.1 The Difficulty of Evaluating Tetris Agents

Designing Tetris agents can be very difficult because of the nature of the game. Evaluating a decent player could take many hours even on a relatively modern computer, making experiments on different algorithms very time consuming. Because of this, researchers adapt their algorithms to reduce the evaluation time by reducing the game length or reducing the number of evaluations.

In reducing the game length, some authors reduced the board size. The player loses faster when the board gets smaller, such as in the case of Bohm et al. [5] where he evaluated Tetris agents on a 6 x 12 game board. Despite reducing the learning time of the algorithm, he discovered that the agent learning process on smaller boards is not the same in the case of larger boards.

Boumanza [2] compared two agents: one agent on a 10 x 20 game board and the other agent on a 6 x 12 game board. The results showed that the first agent learned on a 10 x 20 game board scored better (36.3×10^6) than the other agent (17×10^6). The author drew the same conclusion for smaller board, except for the 10 x 16 board where players performed similarly on the 10 x 20 board. He asserted that some agents are specialized only on specific game boards, and as a result, these agents cannot be generalized on larger boards. The author tried another approach: he increased the frequency of Z and S Tetrominoes, making the game harder because these Tetrominoes lead to producing many holes on the board.

Figure 3summarizes the scores of players learned on smaller games (6 x 12) and players learned on harder games (probability 3/8 for Z and S Tetrominoes) on games of size 10 x 16 and 10 x 20. The results showed that increasing the probability of Z and S Tetrominoes to appear in the game reduces the learning time of the agents.

Game Size	Smaller Games	Larger Games
10 x 16	8.79×10^5	8.7×10^5
10 x 20	17×10^6	36.3×10^6

Figure 3. Learning on Smaller Boards vs. Learning on Larger Boards

2.1.2 Related Works

Tetris agents have been implemented on different algorithms, namely Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), Harmony Search Algorithm (HSA), Genetic Algorithm (GA) and Hand-Coded Algorithms. This section will discuss each algorithm in detail.

2.1.2.1 Particle Swarm Optimization – Improvement on a Tetris Agent (Garrone, 2012)

Particle Swarm Optimization (PSO) is a population-based stochastic metaheuristic optimization technique discovered by Dr. Eberhart and Dr. Kennedy in 1995. The algorithm was inspired by the social behavior of animals such as bird flocking or fish schooling [8]. PSO shares similarity with Evolutionary Algorithms such as the Genetic Algorithm, where the system is initialized with random solutions. The idea of PSO is to iteratively move a candidate solution composed of particles into a hyperspace of parameters in order to improve its values with regard to a given measure.

Garrone [9] applied PSO to optimize the weights of each feature function in Tetris. 25 particles were defined in the study, where particle position comprised a set of weights and the velocities of each particle were the variation between the two iterations. Particle positions were updated by the number of rows returned after one game. Eventually, the swarm reaches an approximate global best position, which is the optimal weights of the feature functions.

Figure 4 shows the comparison of the weights obtained from this study and the weights obtained from another PSO implementation by Islam El-Ashi after performing five iterations.

Features	El-Ashi-computed Weights	Computation Results
Height	-4.5001588	-3.3200740
Number of Eliminated Rows	3.4181268	2.70317569
Number of Row Transitions	-3.2178882	-2.7157289
Number of Column Transitions	-9.3486953	-5.1061407
Number of Holes	-7.8992654	-6.9380080
Well Sums	-3.3855972	-2.4075407

Figure 4. Learning on Smaller Boards vs. Learning on Larger Boards

Figure 5 shows the average number of rows cleared after performing ten games on a standard 10 x 20 board on three algorithms.

Method	Average Cleared Rows
Obtained Results	**13,999,235**
Dellacherie [5]	5,024,731
Islam El-Ashi	16,047,595

Figure 5. Number of Average Cleared Rows on Three Algorithms

The results obtained from study and from Islam El-Ashi were somewhat similar, considering that both used PSO to generate their results. However, the random initialization of the weights of the study is what made the difference in the average cleared rows. The author recommended that further improvement of the evaluation function and the parallelization of the algorithm could be done to improve the results.

2.1.2.2 Ant Colony Optimization – Apply Ant Colony Optimization to Tetris (Chen et al, 2009)

Ant Colony Optimization is another Swarm Intelligence (SI) method, where ants and pheromones are used to solve direction-based problems, such as the Traveling Salesman Problem (TSP).

Chen et al. [24] used ACO to learn the weights of the evaluation function and search an optimal weight-path in the weight graph. Dynamic heuristic was used to prevent premature convergence to local optima. Figure 6 shows the performance of various algorithms with respect to ACO after doing 30 runs.

	Method	Lines	Games
Non-RL	Hand-coded	631,167	N.A.
	GA	586, 103	3000
RL	RRL-KBR	≈50	120
	Policy Iteration	3,183	1500
	LSPI	<3000	≈17
	LP + Bootstrap	4,274	N.A.
	Natural Policy Gradient	≈6,800	≈10,000
	Noisy Cross Entropy + RL	348,895	5000
	ACO + RL, No Heuristic	7,368	4000
	ACO + RL, Dynamic Heuristic	17,586	4000

Figure 6. Performance of Various Algorithms after 30 Runs

The experimental results of the study fared better than most documented RL-Tetris playing programs. However, the Noisy Cross Entropy Algorithm and GA fared significantly better than ACO. The performance of the algorithm could be improved by redesigning the weight graph and utilizing function approximation. In addition, an exponential function could be used instead of a linear function to get better performance.

2.1.2.3 Harmony Search Algorithm – Tetris Agent Optimization Using Harmony Search Algorithm (Romero et al, 2011)

Harmony Search Algorithm (HSA) is a relatively recent meta-heuristic optimization algorithm based on how musicians improve their music developed by Geem et al. [10]. HSA has the ability to discover high performance regions of the solution space in a reasonable amount of time. In addition, HSA imposes fewer mathematics, uses stochastic random searches and creates a new solution vector after considering all existing solution vectors.

Romero et al. [11] applied HSA to Tetris in order to prove a significant relationship between music and the process of looking for an optimal solution. To this end, five harmonies (vector of weights) were tested and ran for two weeks straight. Figure 7 shows the performance of the five harmonies on the 304th cycle (program run).

Harmony	Maximum Number of Cleared Rows (CR)	Total Number of Spawned Pieces (SP)	SP to CP Ratio
X_1	291,087	727,751	2.500115085867
X_2	300,277	750,723	2.50010157288
X_3	337,254	843,168	2.500097849098
X_4	348,047	870,151	2.50009625136
X_5	416,928	1,042,354	2.50008154885

Figure 7. Performance of the Five Harmonies on the 304th Cycle

According to Fahey [5], the theoretical best case for a Tetris game is to able to clear one row using 2.5 numbers of spawned pieces. Furthermore, the Tetris agent gave emphasis on reducing the weight values of the number of holes, wells, column and row transitions, as well as giving importance on the weight value of potential rows. Based on the results, it has shown that HSA is an efficient optimization algorithm that is able to generate the best possible solution. Furthermore, the harmony X_5 had the best harmony since its SP to CP ratio approached the optimum value of 2.5.

2.1.2.4 Genetic Algorithm – Evolutionary AI for Tetris (Shahar et al, 2010)

The Genetic Algorithm is a Swarm-based optimization algorithm (SOA) which is based on natural selection and genetic recombination. The algorithm works by choosing solutions from the current population and applying genetic operators such as mutation and crossover to create a new population. GA exploits historical information to speculate on new search areas with improved performance [12]. When applied to optimization problems, GA has the advantage of performing global search.

Shahar et al. [13] examined an evolutionary approach to the video game Tetris using Genetic Algorithms to evolve a set of optimal weights for features detectors in a state space search of the Tetris game. Choosing 14 genes (feature functions), the weights were generated randomly and the Tetris agent started at generation 0. The genes are updated every time the game ends, and are applied genetic operators (reproduction, mutation and crossover) in order to improve the weights.

The Tetris agent was assessed on three tests: 100 Line Limit (force the game to end after clearing 100 lines), 1000 Line Limit (force the game to end after clearing 1000 lines) and Unlimited (No limit). The first two tests made the program rapidly approached the maximum-possible score, and fluctuated slightly on a near-flat line as mutations appeared. The third test has a relatively small dataset, because the program could not get past 3 evolutions in a reasonable amount of time. Despite this, the agent played well enough that it lasted a significant amount of time. The author stressed that the third test should be explored more in order to test how well the agent plays without restrictions.

2.1.2.5 Hand-Coded Algorithms

Hand-Coded Algorithms make use of the knowledge of the individual and testing the program through trial and error methods. So far, the current best one-piece Tetris agent is due to Dellacherie [5] and was tuned by hand. He came up with important feature functions and manually adjusted their weights. Surprisingly, his manually-tuned one-piece agent outperforms existing one-piece agents in the literature, even though their weights were automatically adjusted.

After conducting 56 games in the original Tetris game, his algorithm completed an average score of about 660,000 lines per game. His evaluation function is linear combination of the ff:

– (Landing Height) + (Eroded Piece Cells) – (Row Transitions) – (Column Transitions) – 4 x (Holes) – (Cumulative Wells)

In the same manner, Fahey [5] created a two-piece controller of which the weights were also tuned by hand. He reported a game score of 7,200,000 in only one game and only took a week to get the results.

2.2 The Bees Algorithm

The Bees Algorithm (BA) is a metaheuristic Swarm Intelligence approach that belongs to the class of nature-inspired algorithms. It was developed by Pham et al. in 2005; the inspiration for the algorithm came from a number of biological and natural processes [10]. This search technique utilizes a resemblance between the way in which bees in nature search for food, and the way in which optimization algorithms search for an optimal solution in difficult combinatorial optimization problems.

The BA mimics the food foraging behavior of honey bees residing in a colony. The foraging process starts when the bee colony has send scout bees to collect nectar from flower patches relative to the amount of food at each patch. Upon returning to the hive, the scout bees communicate with the rest of the bees through a waggle dance. This ritualistic dance conveys three important information: the direction of the flower patch, the distance from the hive to the patch and the quality rating (or fitness) of the food source. The waggle dance is essential, because the information supports the colony to send its scout bees to flower patches accurately, without using guides or maps. In addition, more follower bees are sent to more promising patches. This type of communication allows the colony to gather food quickly and efficiently.

2.2.1 Applications of Bees Algorithm

BA has been successfully applied to various engineering and management optimization problems. Moreover, it has been applied in some problems like multi-objective optimization [14], neural network training for wood defects [15], manufacturing cell formation [16], job scheduling for a machine [17], data clustering [18], optimizing the design of mechanical components [19], image analysis [20] and supply chain optimization [21].

2.2.2 Related Works

2.2.2.1 The Bees Algorithm - A Novel Tool for Complex Optimization Problems

The work of Pham et al. [22] provided results obtained for a number of benchmark problems signifying the efficiency and robustness of BA, while being compared to two metaheuristics: GA and ACO.

Benchmark Function	GA		ACO		BA	
	Mean No. of Evaluations (Number of Iterations)	Success %	Mean No. of Evaluations (Number of Iterations)	Success %	**Mean No. of Evaluations (Number of Iterations)**	**Success %**
De Jong	10160	100	6000	100	**868**	**100**
Goldstein & Price	5662	100	5330	100	**999**	**100**
Branin	7325	100	1936	100	**1657**	**100**
Martin & Gaddy	2844	100	1688	100	**526**	**100**
Rosenbrock	10212	100	6842	100	**631**	**100**
Hyper Sphere	15468	100	22050	100	**7113**	**100**
Greiwangk	200000	100	50000	100	**1847**	**100**

Figure 8. Test Functions: GA, ACO and BA [22]

The results were quite promising, as BA revealed significant robustness, generating a 100% success percentage in all test functions. Based on De Jong's test function, BA was 120 times faster than ACO and 207 times faster than GA. With Goldstein and Price's test function, BA was 5 times faster than ACO and GA. Through Branin's test function, a 15% improvement was noted in BA compared to ACO, and 77% improvement when compared to GA. Both Martin-Gaddy and Rosenbrock's test functions bought a 100% success percentage to BA, which is a significant improvement over GA and ACO. In the Hyper Sphere test function, BA required half the mean number of evaluations compared to GA and one-third of that required for ACO. Lastly, evaluating Greiwangk's test function revealed BA to be 10 times faster than GA and 25 times faster than ACO. In short, BA outperformed ACO and GA in terms of speed of optimization and accuracy of the results obtained, and can be considered a tool for complex optimization problems.

2.2.2.2 Forming Student Groups Using Bees Algorithm

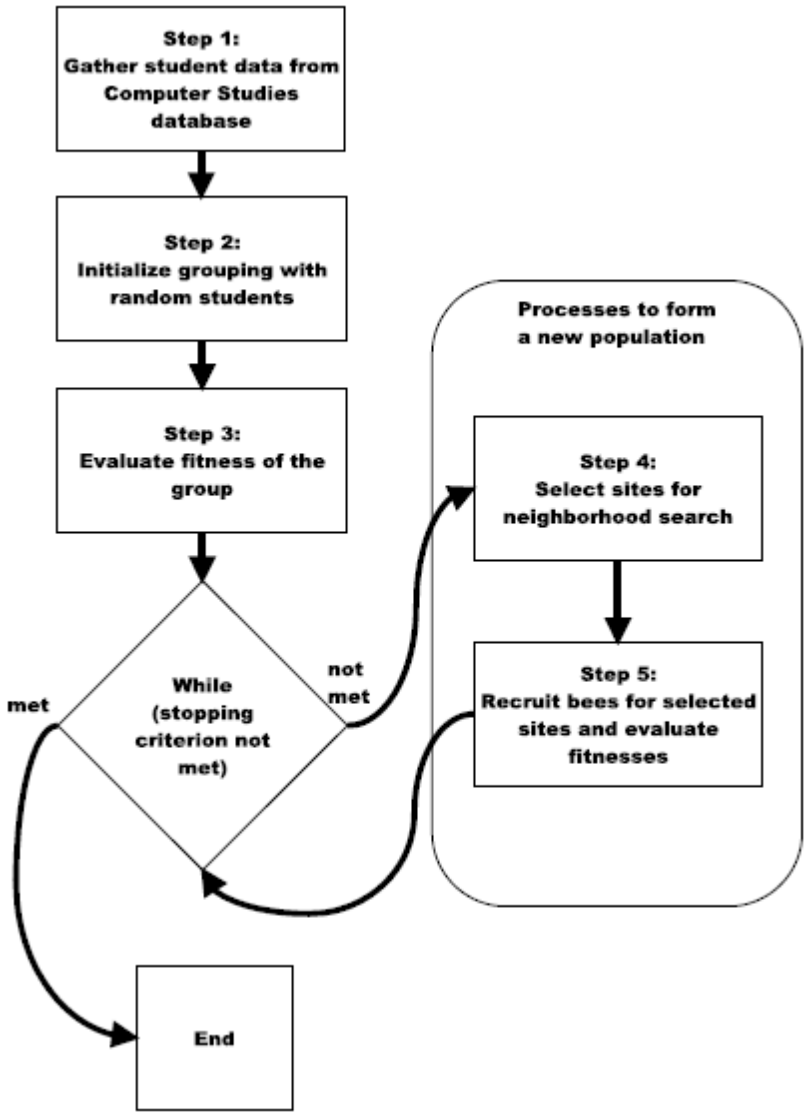

Figure 9. The Conceptual Framework of Ablazo et al. [23]

Ablazo et al. [23] utilized BA and an objective function to form groups of three students each, where grades on major subjects were the only bases of the students' grouping. With all objectives accomplished, the results delivered a good measure of heterogeneity and fitness. However, the proponents believed that grades are not the sole basis for forming groups, as what they identified in their data gathering. They recommended finding a new mathematical approach to cater two-student group formations, as well as addition other criteria to consider such as minor subjects, personal preferences, etc.

2.3 Theoretical Framework

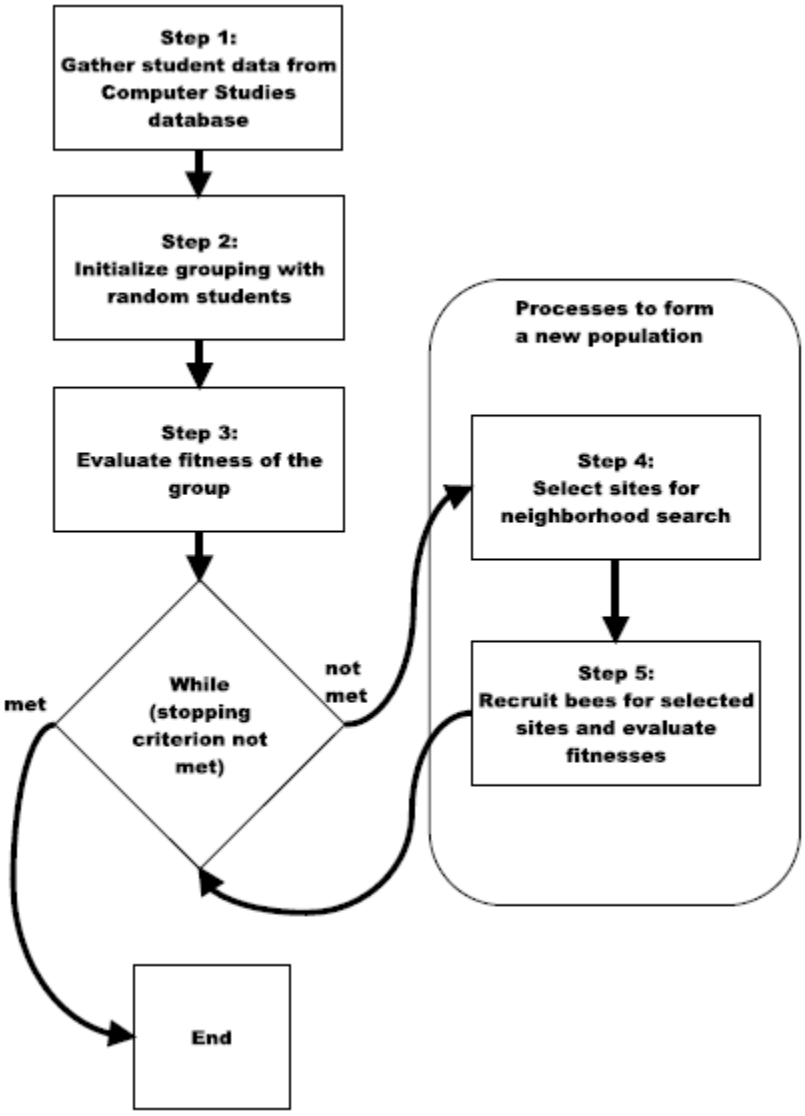

Figure 10. Theoretical Framework by Ablazo et al. [23]

3. RESEARCH DESIGN AND METHODOLOGY

3.1 Conceptual Framework

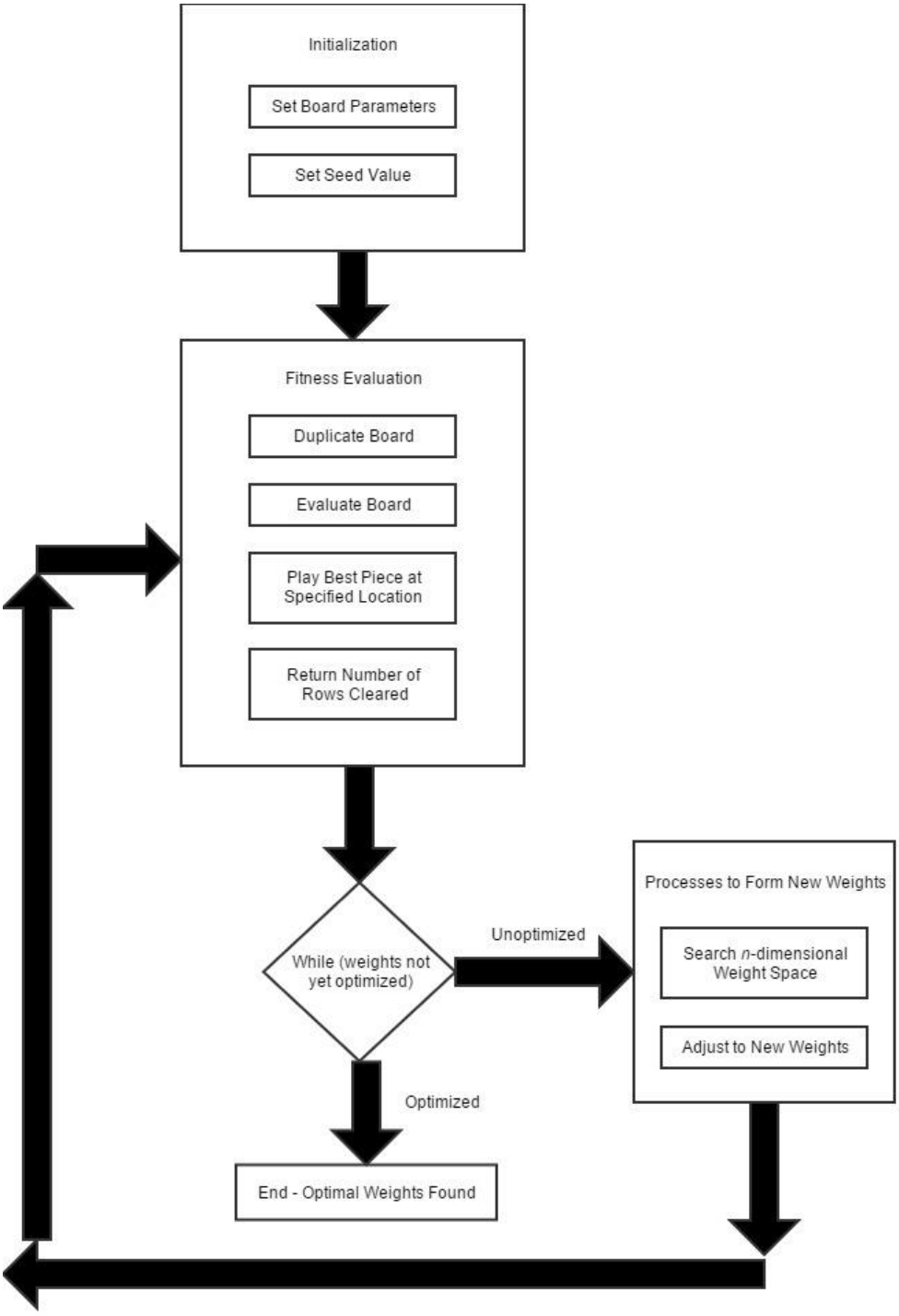

Figure 11. Conceptual Framework

3.1.1 Set Board Parameters

The first step is to initialize the Tetris board with a specific dimension. Standard Tetris boards have a dimension of 10 by 20, but can be adjusted to reduce the length of the game.

3.1.2 Set Seed Value

The second step is to set an arbitrary seed value before playing the game. The purpose of setting the seed value is to initialize the same sequence of Tetrominoes in every game.

3.1.3 Duplicate Board

The third step is to start the game and duplicate the board in order to evaluate the piece.

3.1.4 Evaluate Board

The fourth step is to drop the Tetromino on the duplicated board. When dropping the Tetromino does not result to a game termination, the board is assessed through an evaluation function, where the board is given a merit based on the number of features selected and the weights assigned per feature. If the resulting evaluation leads to a greater merit than those of previous evaluations, then the best evaluation is updated, as well as the best orientation and column placements of the Tetromino. The different orientations of the Tetromino are evaluated as well by duplicating the original board, as in Step 3.1.3.

3.1.5 Play Best Piece at Specified Location

The fifth step is to drop the Tetromino on the actual board, using the best orientationand column placements acquired from Step 3.1.4. Steps 3.1.3 to 3.1.4 are repeated for the next pieces that are dropped in the game board.

3.1.6 Return Number of Rows Cleared

The sixth step is to return the number of rows cleared by the game when a game termination occurs.

3.1.7 Search n-dimensional Weight Space

The seventh step is to search the n-dimensional weight space if the weights are not yet optimized, where n corresponds to the number of features set in the game. This is akin to bees searching the neighborhood for optimal sites.

3.1.8 Adjust to New Weights

The eight step is to select the weights obtained from the n-dimensional weight space as the current best weights of the game. Afterwards, the game is played again using the new weights, repeating Steps 3.1.3 to 3.1.8. Searches in the most optimal weight space continue on nearby spaces in order to fetch other optimal weights faster.

3.1.9 End – Optimal Weights Found

Once the optimal weights have been found after playing the game a number of times, the testing is stopped and an optimal set of weights is obtained. The optimal weights differ between board dimensions and testing of different board dimensions should be conducted in order to find their own optimal weights.

3.2 Methodology

3.2.1 Define Program Parameters

The thesis is done by first defining the program parameters, such as board width, board height, Tetromino drop speed and the number of rows cleared.

The board width, board height and Tetromino drop speed can be adjusted in order to try out different results. The number of rows cleared is a display and will reset for every game iteration. In addition, a feature called Hardcore Mode allows faster results, but may lead to a slower system. Figure 12 shows the program parameters in the front-end program.

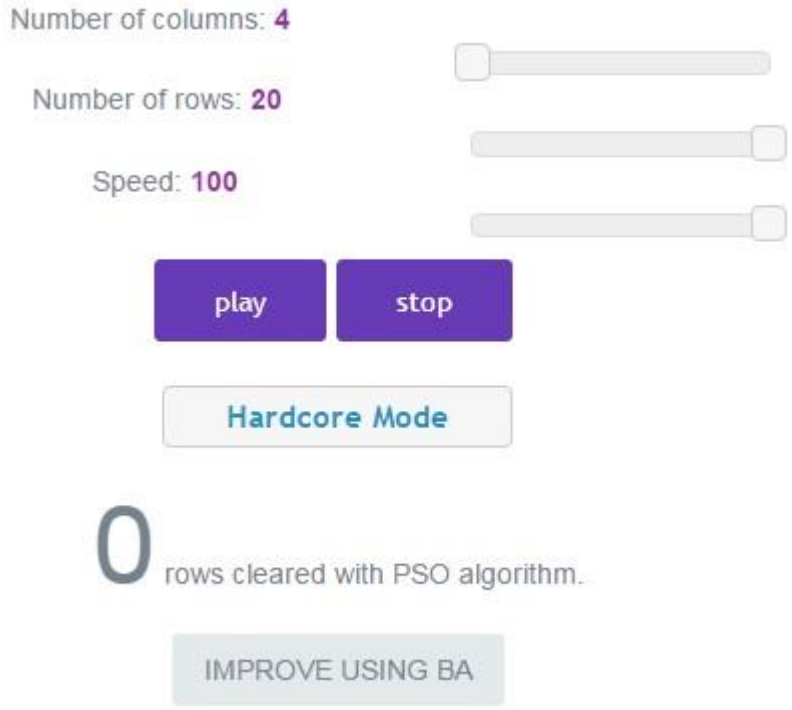

Figure 12. Program Parameters

3.2.2 Develop Criteria for Evaluation

The evaluation function is created using a number of features. In the study, six features are used: the landing height, rows removed, row transitions, column transitions, number of holes and well sums. The evaluation function returns a linear summation of these features with their corresponding weights, which is then assigned to a board (merit) to highlight its significance.

The details of the feature functions used are as follows:

Landing height refers to the height at which the last Tetromino was placed.

```
function GetLandingHeight(last_move, board) {
    return last_move.landing_height + ((last_move.piece.length - 1) / 2);
}
```

Figure 13. Landing Height

Removed rows are rows cleared by the last Tetromino, in order to arrive at the new board configuration.

```
for (i = 0; i < piece.length; i++) {
    if (board[placementRow + i] == this.FULLROW) {
        board.splice(placementRow + i, 1);
        // Add an empty row on top.
        board.push(0);
        // Since we have decreased the number of rows by one, we need to adjust
        // the index accordingly.
        i--;
        rowsRemoved++;
    }
}
```

Figure 14. Removed Rows

Row transitions are the sum of all occupied or unoccupied horizontal transitions.

```
function GetRowTransitions(board, num_columns) {
    var transitions = 0;
    var last_bit = 1;

    for (var i = 0; i < board.length; ++i) {
        var row = board[i];

        for (var j = 0; j < num_columns; ++j) {
            var bit = (row >> j) & 1;

            if (bit != last_bit) {
                ++transitions;
            }

            last_bit = bit;
        }

        if (bit == 0) {
            ++transitions;
        }
        last_bit = 1;
    }
    return transitions;
}
```

Figure 15. Row Transitions

Column transitions are the same with row transitions, but only counts vertical transitions.

```
function GetColumnTransitions(board, num_columns) {
    var transitions = 0;
    var last_bit = 1;

    for (var i = 0; i < num_columns; ++i) {
        for (var j = 0; j < board.length; ++j) {
            var row = board[j];
            var bit = (row >> i) & 1;

            if (bit != last_bit) {
                ++transitions;
            }

            last_bit = bit;
        }

        last_bit = 1;
    }

    return transitions;
}
```

Figure 16. Column Transitions

The number of holes are the number of gaps with at least one occupied cell above them.

```
function GetNumberOfHoles(board, num_columns) {
    var holes = 0;
    var row_holes = 0x0000;
    var previous_row = board[board.length - 1];

    for (var i = board.length - 2; i >= 0; --i) {
        row_holes = ~board[i] & (previous_row | row_holes);

        for (var j = 0; j < num_columns; ++j) {
            holes += ((row_holes >> j) & 1);
        }

        previous_row = board[i];
    }

    return holes;
}
```

Figure 17. Number of Holes

Well sums are the sum of all wells on the game board. It checks for wells in the leftmost and rightmost columns of the board.

```
function GetWellSums(board, num_columns) {
    var well_sums = 0;

    // Check for well cells in the "inner columns" of the board.
    // "Inner columns" are the columns that aren't touching the edge of the board.
    for (var i = 1; i < num_columns - 1; ++i) {
        for (var j = board.length - 1; j >= 0; --j) {
            if ((((board[j] >> i) & 1) == 0) &&
                (((board[j] >> (i - 1)) & 1) == 1) &&
                (((board[j] >> (i + 1)) & 1) == 1)) {

                // Found well cell, count it + the number of empty cells below it.
                ++well_sums;

                for (var k = j - 1; k >= 0; --k) {
                    if (((board[k] >> i) & 1) == 0) {
                        ++well_sums;
                    } else {
                        break;
                    }
                }
            }
        }
    }
    for (var j = board.length - 1; j >= 0; --j) {
        if ((((board[j] >> 0) & 1) == 0) &&
            (((board[j] >> (0 + 1)) & 1) == 1)) {

            // Found well cell, count it + the number of empty cells below it.
            ++well_sums;

            for (var k = j - 1; k >= 0; --k) {
                if (((board[k] >> 0) & 1) == 0) {
                    ++well_sums;
                } else {
                    break;
                }
            }
        }
    }
```

```
    // Check for well cells in the rightmost column of the board.
    for (var j = board.length - 1; j >= 0; --j) {
        if ((((board[j] >> (num_columns - 1)) & 1) == 0) &&
            (((board[j] >> (num_columns - 2)) & 1) == 1)) {
            // Found well cell, count it + the number of empty cells below it.

            ++well_sums;
            for (var k = j - 1; k >= 0; --k) {
                if (((board[k] >> (num_columns - 1)) & 1) == 0) {
                    ++well_sums;
                } else {
                    break;
                }
            }
        }
    }

    return well_sums;
}
```

Figure 18. Well Sums

3.2.3 Implement Bees Algorithm through a Tetris Program

After defining the program parameters and the evaluation criteria, a front-end interface will be created that includes Bees Algorithm and the developed criteria as shown in Figure 19.

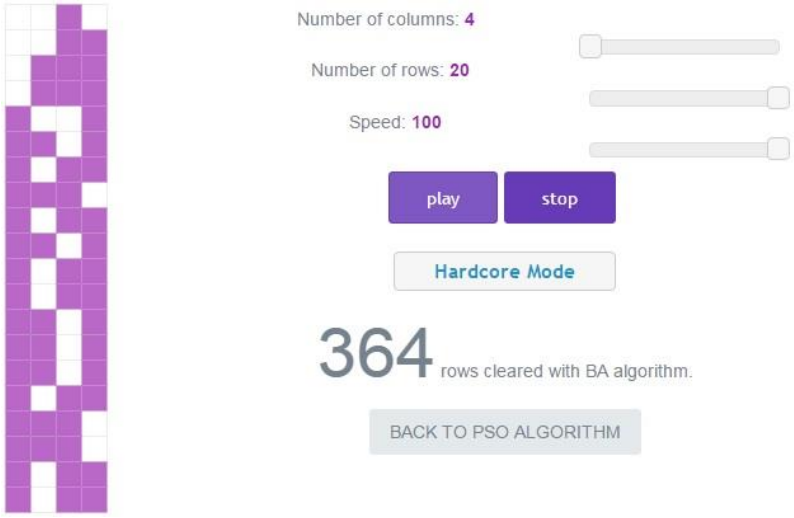

Figure 19. Front-End Interface

3.2.4 Test and Evaluate Tetris Program

Once the front-end program has been developed, it is run for a number of times in order to obtain the best weights for a given board configuration. The weights of the features are considered to be the best when the program has obtained the highest number of rows cleared.

4. RESULTS AND DISCUSSION

After conducting 200 tests, the optimal weights for each feature function were obtained on different boards, as shown in Figure 20.

Features	Weights (PSO) – Computed by El-Ashi	Weights (BA) – 4x20	Weights (BA) – 5x20	Weights (BA) – 6x20	Weights (BA) – 7x20	Weights (BA) – 8x20
Height	-4.5001588	-5.0	-7.0	-6.0	-6.0	-2.0
Number of Eliminated Rows	3.4181268	10.0	5.0	5.0	5.0	3.0
Number of Row Transitions	-3.2178882	-9.0	-4.0	-2.0	-2.0	-1.0
Number of Column Transitions	-9.3486953	-7.0	-7.0	-10.0	-10.0	-1.0
Number of Holes	-7.8992654	-7.0	-9.0	-10.0	-10.0	-8.0
Well Sums	-3.3855972	-3.0	-5.0	-5.0	-5.0	-2.0

Figure 20.Weight Comparison between PSO and BA Weights

The performance of a Tetris agent can be evaluated by the number of cleared rows. Testing is performed by first using the weights from El-Ash, then using the weights generated by BA for every board. Figure 21 shows the number of rows cleared by *BeeTris* when run for 200 times each on different boards.

Board Size	BA	PSO
4 x 20	364	87
5 x 20	860	317
6 x 20	2653	341
7 x 20	4589	1748
8 x 20	56007	11997

Figure 21. Number of Rows Cleared by BA and PSO on Different Board Dimensions

Figure 22 shows the graph of the number of rows cleared between BA and PSO in different board dimensions more accurately.

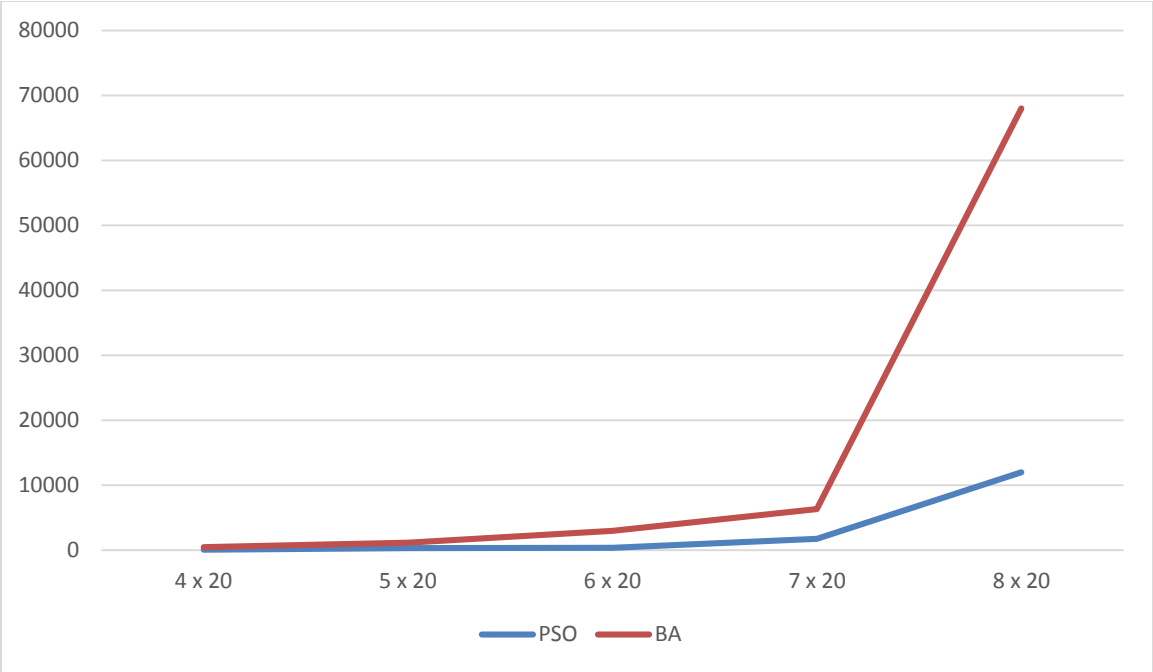

Figure 22. Graph on Number of Rows Cleared

From Figures 21 and 22, BA performed better than PSO. The difference between the graphs could be seen at the 7 x 20 board column where BA leaned higher than PSO.

5. CONCLUSIONS AND RECOMMENDATIONS

5.1 Conclusions

Computational results reveal that BA is an efficient optimization algorithm. *BeeTetris,* the Tetris agent used in the study, is able to generate a set of optimal weights within 200 runs. The optimal weights acquired were used to clear the maximum number of rows possible a given board dimension. As a stochastic metaheuristic algorithm, BA employs some degree of randomization which made it perform better than the PSO-generated weights from El-Ashi (He did not use randomization in generating the optimal weights). Also, the weights for the feature functions are not universal; each board dimension has a specific set of weight values to produce the optimal number of rows cleared. In addition, increasing the weight values of the number of holes and reducing the number of wells provide better results.

5.2 Recommendations

The study conducted test runs on only five game boards: 4 x 20, 5 x 20, 6 x 20, 7 x 20 and 8 x 20. Because of time constraints, testing on bigger boards (such as 9 x 20, 10 x 20, etc.) should be conducted, since these boards take a very long time to complete. In addition, a mathematical equation could be created to predict or compute cleared rows within a given set of weights and game board parameters. Lastly, a comparison on different Swarm Intelligence (SI) algorithms other than Particle Swarm Optimization (PSO) should be done to determine which algorithm is superior in terms of the number of rows cleared per game.

REFERENCES

[1] Burgiel, Heidi. How to lose at Tetris. *Mathematical Gazette*, 81 (1997), 194-200.

[2] Boumaza, Amine. How to design good Tetris players.

[3] Thiery, Christophe and Scherrer, Bruno. Building controllers for Tetris. *International Computer Games Association Journal*, 32 (2009), 3-11.

[4] Demaine, Erik D, Hohenberger, Susan, and Liben-Nowell, David. Tetris is hard, even to approximate. In *Computing and combinatorics*. Springer, 2003.

[5] Fahey, Colin. Tetris. *Colin Fahey* (2003).

[6] B, K, and Mandl, Stefan. An evolutionary approach to tetris. In *The Sixth Metaheuristics International Conference (MIC2005)* (2005), 5.

[7] Langenhoven, Leo, Van Heerden, Willem S, and Engelbrecht, Andries P. Swarm tetris: Applying particle swarm optimization to tetris. In *Evolutionary Computation (CEC), 2010 IEEE Congress on* (2010), 1-8.

[8] Hu, Xiaohui. PSO tutorial. *URL: http://www.swarmintelligence.org/tutorials.php* (2006).

[9] Garrone, Julie. Improvement on a Tetris agent. URL*: https://juliegarrone.wordpress.com/skills-mini-projects/* (2012).

[10]Geem, Zong Woo, Kim, Joong Hoon, and Loganathan, GV. A new heuristic optimization algorithm: harmony search. *Simulation*, 76, 2 (2001), 60-68.

[11] Romero, Victor M, Tomes, Leonel L, and Yusiong, John Paul T. Tetris Agent Optimization Using Harmony Search Algorithm. *International Journal of Computer Science Issues*, 8, 1 (2011), 22-31.

[12] Goldberg, David E and others. *Genetic algorithms in search optimization and machine learning*. Addison-wesley Reading Menlo Park, 1989.

[13] Shahar, Elad and West, Ross. Evolutionary AI for Tetris.

[14] Pham, DT and Ghanbarzadeh, Afshin. Multi-objective optimisation using the bees algorithm. In *Proceedings of IPROMS 2007 Conference* (2007).

[15] Pham, Duc Truong, Soroka, Anthony J, Ghanbarzadeh, Afshin, Koc, Ebubekir, Otri, Sameh, and Packianather, Michael. Optimising neural networks for identification of wood defects using the bees algorithm. In *Industrial Informatics, 2006 IEEE International Conference on* (2006), 1346-1351.

[16] Pham, Duc Truong, Afify, Ashraf, and Koc, Ebubekir. Manufacturing cell formation using the Bees Algorithm. In *Innovative Production Machines and Systems Virtual Conference, Cardiff, UK* (2007).

[17] Pham, DT, Koc, E, Lee, JY, and Phrueksanant, J. Using the bees algorithm to schedule jobs for a machine. In *Proceedings Eighth International Conference on Laser Metrology, CMM and Machine Tool Performance, LAMDAMAP, Euspen, UK, Cardiff* (2007), 430-439.

[18]Pham, DT, Otri, S, Afify, A, Mahmuddin, Massudi, and Al-Jabbouli, H. Data clustering using the bees algorithm. In *Proceedings of 40th CIRP international manufacturing systems seminar* (2007).

[19]Pham, DT, Soroka, AJ, Koc, E, Ghanbarzadeh, A, and Otri, S. Some applications of the bees algorithm in engineering design and manufacture. In *Proceedings of international conference on manufacturing automation (ICMA 2007), Singapore* (2007).

[20]Yuce, Baris. *Novel computational technique for determining depth using the Bees Algorithm and blind image deconvolution*. 2012.

[21]Mastrocinque, Ernesto, Yuce, Baris, Lambiase, Alfredo, and Packianather, Michael S. A multi-objective optimisation for supply chain network using the bees algorithm. *Int. J. Eng. Bus. Manage*, 5 (2013), 1-11.

[22]Pham, DT, Ghanbarzadeh, A, Koc, E, Otri, S, Rahim, S, and Zaidi, M. The Bees Algorithm--A Novel Tool for Complex Optimisation. In *Intelligent Production Machines and Systems-2nd I* PROMS Virtual International Conference 3-14 July 2006* (2011), 454.

[23] Ablazo, A.C., Bonga, E.J.R., Viscayno, V.C.B. 2012. Forming student groups using Bees Algorithm. Ateneo De Davao University. School Of Arts And Sciences, Davao City

[24] Chen, Xingguo, Wang, Hao, Wang, Weiwei, Shi, Yinghuan, and Gao, Yang. Apply ant colony optimization to tetris. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (2009), 1741-1742.